Exercices — asyncio

Exercice 1 — Sync vs async : mesurer la différence

Vous avez ces trois fonctions synchrones qui simulent des appels réseau lents :

```
import time

def fetch_users():
    time.sleep(2)
    return ["alice", "bob", "charlie"]

def fetch_orders():
    time.sleep(3)
    return ["order_1", "order_2"]

def fetch_inventory():
    time.sleep(1)
    return ["item_A", "item_B", "item_C"]
```

1. Écrivez une version synchrone qui appelle les trois fonctions l'une après l'autre et mesure le temps total.
2. Convertissez les trois fonctions en coroutines (`asyncio.sleep` à la place de `time.sleep`).
3. Écrivez une version asynchrone qui les lance toutes les trois en même temps avec `asyncio.gather()` et mesure le temps total.
4. Comparez les deux durées et expliquez pourquoi elles diffèrent.

Résultat attendu pour la version async : environ 3 secondes (le maximum des trois), pas 6 (la somme).

Exercice 2 — Téléchargeur concurrent avec progression

Simulez le téléchargement de 5 fichiers en parallèle. Chaque fichier a une taille aléatoire entre 1 et 5 secondes de téléchargement.

Contraintes :

- Utiliser `asyncio.create_task()` pour lancer les téléchargements
- Afficher un message dès qu'un fichier est terminé, dans l'ordre d'achèvement (pas l'ordre de lancement) — utilisez `asyncio.as_completed()`
- Afficher à la fin le temps total et le nom du fichier le plus long à télécharger

Exemple de sortie attendue :

```
Début des téléchargements...
✓ fichier_3.zip terminé (1.2s)
✓ fichier_1.zip terminé (2.4s)
✓ fichier_5.zip terminé (3.1s)
```

```
✓ fichier_2.zip terminé (4.0s)
✓ fichier_4.zip terminé (4.8s)
Temps total : 4.8s — fichier le plus lent : fichier_4.zip
```

Exercice 3 — Timeout sur des tâches réseau

Vous interrogez 4 serveurs. Certains répondent vite, d'autres sont lents ou ne répondent jamais. Simulez ce comportement :

```
SERVEURS = {
    "serveur_A": 0.5,
    "serveur_B": 2.0,
    "serveur_C": 5.0,   # trop lent
    "serveur_D": 1.2,
}
```

Écrivez une coroutine `interroger(nom, delai)` qui simule l'appel au serveur. Ensuite, dans `main()` :

1. Appliquez un timeout de 2 secondes à chaque appel avec `asyncio.wait_for()`
2. Si un serveur dépasse le timeout, affichez un message d'avertissement et continuez avec les autres
3. Affichez à la fin un résumé : serveurs ayant répondu / serveurs en timeout

Exercice 4 — Pipeline producteur / consommateur

Vous avez une liste de 6 URLs d'API à interroger (simulées). Chaque URL retourne une liste de résultats qui doit ensuite être analysée.

Mettez en place un pipeline avec `asyncio.Queue` :

- 1 producteur qui "récupère" les données de chaque URL (délai aléatoire 0.5–2s)
- 2 consommateurs qui analysent les données en parallèle dès qu'elles arrivent (délai fixe 1s par analyse)
- Le producteur signale la fin avec des sentinelles `None`
- Affichez le temps total et comparez-le avec ce que prendrait un pipeline séquentiel

L'objectif : montrer que 2 consommateurs en parallèle traitent plus vite qu'un seul.

Exercice 5 — Détecteur de services actifs

Écrivez un script `check_services.py` qui vérifie si une liste de services locaux répond, en simulant des checks de disponibilité :

```
SERVICES = [
    {"nom": "base de données", "delai": 0.3},
    {"nom": "cache Redis",     "delai": 0.1},
    {"nom": "API externe",     "delai": 3.5},  # timeout
    {"nom": "serveur mail",    "delai": 0.8},
```

```
    {"nom": "CDN",              "delai": 1.5},
]
```

Contraintes :

- Tous les checks se lancent en même temps
- Timeout de 2 secondes par service
- Résultat affiché dès qu'un check est terminé (pas à la fin de tous)
- Un service en timeout est marqué `HORS LIGNE`
- À la fin : nombre de services en ligne / hors ligne, et temps total

Ce script doit ressembler à un vrai outil de monitoring : propre, lisible, utilisable en production.